LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

A Syntactic Approach to Program Transformation

Computation Structures Group Memo 322
10 December 1990

Zena M. Ariola
Arvind

This report describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for this work has been provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-89-J-1988 (MIT) and N0039-88-C-0163 (Harvard).

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Contents

A Syntactic Approach to Program Transformations

Zena M. Ariola
Aiken Computational Laboratory
Harvard University

Arvind
Laboratory for Computer Science
Massachusetts Institute of Technology

December 10, 1990

Abstract

Kid, a language for expressing compiler optimizations for functional languages is introduced. The language is λ-calculus based but treats let-blocks as first class objects. The language goes beyond λ-calculus by including I-structures which are essential to express efficient translations of list and array comprehensions. A calculus and a interpreter for Kid are developed. Many commonly known program transformations are also presented. A *partial evaluator* for Kid is developed and a notion of *correctness* of Kid transformations based on the syntactic structure of terms and *printable answers* is presented.

Keywords and phrases: λ-calculus, Term Rewriting Systems, Contextual Reduction Systems, Confluence, Optimizations, Correctness.

1 Introduction

Most compilers have a machine independent phase followed by one or more back-ends that generate code for specific machines. Compiler optimizations in the machine-independent phase have long term pay offs and are consequently very desirable. Clean semantics of declarative languages offer great opportunities for machine-independent optimizations and analyses. It is, therefore, desirable to have an intermediate language with proper semantics where optimizations can be expressed as source-to-source transformations. Such an intermediate language must have the capability to express operational concerns. The idea of having a calculus that reflects what happens operationally goes back to the work of Plotkin [14].

Experience of many researchers has shown that the λ-calculus with constants is not adequate as an intermediate language for compilers of functional languages. One of its primary deficiency is the inability to capture the sharing of subexpressions. Avoiding repeated evaluation of an expression is a central concern in implementations of non-strict functional languages. Not surprisingly graph reduction [18], [16]

[10], has been one of the popular ways of implementing functional languages, but it is only recently that people have investigated suitable calculus for graph rewriting [5], [11], and [6].

Another major short coming of λ-calculus, or any other purely functional language, is the inability to express "efficient" translations of list and array comprehensions. The usual translations [15] are unnecessarily sequential and elude to other implementation tricks to avoid construction of intermediate data structures. We believe I-structures are essential to express realistic translation of many constructs in a language such as Haskell [9] and Miranda [17].

We, therefore, introduce Kid, a programming language, and its associated calculus. Kid is essentially the λ-calculus plus the block construct. However, the "block" is not treated merely as syntactic sugar for applications. Kid also contains I-structures [3] which take us beyond the realm of pure functional languages. Kid embodies the novel idea of *multiple values* as well. Kid is a much more refined version of the language P-TAC presented earlier by the authors [1].

Having formalized the calculus, what we want to show is that questions related to correctness of optimizations can be precisely formalized. Usually questions related to correctness take us in the realm of model theories, because, equality of terms can not be derived from the formal system. As an example, consider a very simple rule, called alg,

$$\mathsf{And}\ (\mathsf{True},\ X) \longrightarrow X$$

A notion of α-equivalence will certainly not be enough to show that the above rule is safe. However, by defining different notions of equivalence between terms in a syntactic domain, we will be able to relate any two term M and N such that $M \xrightarrow{\mathsf{alg}} N$. As we will see, these equalities are based on the answer associated to a term, where the answer is the maximal information that a term can produce. Informally, we will consider an optimization to be correct if for any two programs M and N, where N is the optimized version of M, the answer of N is more defined than the answer of M. If the optimized program produces more information, we are still willing to call that optimization correct. What we are after is a complete *syntactic characterization of interesting program equivalences.*

In Section 2 we introduce Kid and informally explain the translation of list comprehensions using I-structures. In Section 3 we introduce Contextual Reduction Systems (CRS's) and give basic rules for CRS's. A notion of canonical terms is also developed. Kid rewrite rules and a calculus for Kid are presented in Section 4. We also present the notion of printable answers associated with Kid terms and give a normalizing interpreter that produces these answers for any Kid term. Section 5.1 contains optimizations and a partial evaluator for Kid. The correctness of Kid transformations is presented next. Finally we give our thoughts on future work.

MV	$\in$	Multiple Variable
SE	$\in$	Simple Expression
PFi_m	$\in$	Primitive Function with i arguments and m outputs
$\mathit{Ap\_E}_m$	$\in$	Applicative Expression with m outputs
$\mathit{Case\_E}_m$	$\in$	Case Expression with m outputs
$\mathit{Loop\_E}_m$	$\in$	Loop Expression with m outputs
$\mathit{lambda\_E}$	$\in$	Lambda Expression
$\mathit{E\_m}$	$\in$	Expression with m outputs
$\mathit{Variable}$	$::=$	$x \mid y \mid z \mid \cdots \mid a \mid b \mid \cdots \mid f \mid \cdots \mid x_1 \mid \cdots$
MV_m	$::=$	$\underbrace{\mathit{Variable}, \cdots, \mathit{Variable}}_{m}$
$\mathit{Constant}$	$::=$	$\mathit{Integer} \mid \mathit{Boolean} \mid () \mid (\mathsf{Not}) \mid \cdots \mid (+) \mid (-) \mid \cdots$
		$\mid \mathsf{Nil} \mid \mathsf{Make\_array} \mid \mathsf{Error} \mid \top$
SE	$::=$	$\mathit{Variable} \mid \mathit{Constant}$
SE_m	$::=$	$\underbrace{\mathit{SE}, \cdots, \mathit{SE}}_{m} \mid \mathit{SE}, \mathit{SE}_{m-1} \mid \mathit{SE}, \mathit{SE}, \mathit{SE}_{m-2} \mid \cdots$
$\mathit{PF1}_1$	$::=$	$\mathsf{Negate} \mid \mathsf{Not} \mid \mathsf{Bounds} \mid \mathsf{I\_array} \mid \mathsf{Open\_cons} \mid \mathsf{Cons\_1} \mid \mathsf{Cons\_2}$
$\mathit{PF1}_m$	$::=$	$\mathsf{Detuple}_\mathsf{m}$
$\mathit{PF2}_1$	$::=$	$+ \mid - \mid * \mid \cdots \mid \mathsf{Equal?} \mid \mathsf{Eq?} \mid < \mid \cdots \mid \mathsf{And} \mid \mathsf{Cons} \mid \mathsf{Apply} \mid \mathsf{Apply}_{pe} \mid \mathsf{P\_select} \mid \mathsf{Make\_tuple}_2$
PFN_1	$::=$	$\mathsf{Make\_tuple}_\mathsf{n} \mid \mathsf{Apply}_{\mathsf{n}-2}$
$\mathit{Ap\_E}_m$	$::=$	$\mathsf{Ap}_{\mathsf{n,m}}\ (\mathit{SE}_{n+1})$
$\mathit{Case\_E}_m$	$::=$	$\mathsf{Bool\_case}_\mathsf{m}\ (\mathit{SE},\ E_m,\ E_m) \mid \mathsf{List\_case}_\mathsf{m}\ (\mathit{SE},\ E_m,\ E_m)$
$\mathit{Loop\_E}_m$	$::=$	$\mathsf{WLoop}_\mathsf{m}\ (\mathit{SE}_{m+3}) \mid \mathsf{FLoop}_\mathsf{m}\ (\mathit{SE}_{m+4})$
$\mathit{lambda\_E}$	$::=$	$\lambda_{\mathsf{n,m}}\ (\mathit{MV}_n) \,.\, (E_m) \mid \underline{\lambda_{\mathsf{n,m}}}\ (\mathit{MV}_n) \,.\, (E_m)$
E_1	$::=$	$\mathit{SE}_1 \mid \mathit{PF1}_1\ (\mathit{SE}) \mid \mathit{PF2}_1\ (\mathit{SE}_2) \mid \mathit{PFN}_1\ (\mathit{SE}_n) \mid \mathit{Ap\_E}_1 \mid \mathit{Case\_E}_1 \mid \mathit{Loop\_E}_1 \mid \mathit{lambda\_E} \mid \mathit{Block}_1$
E_m	$::=$	$\mathit{SE}_m \mid \mathit{PF1}_m\ (\mathit{SE}) \mid \mathit{Ap\_E}_m \mid \mathit{Case\_E}_m \mid \mathit{Loop\_E}_m \mid \mathit{Block}_m$
Block_m	$::=$	$\{\mathsf{m}\ [\mathit{Statement};]^*\ \mathsf{In}\ \mathit{SE}_m\}$
$\mathit{Statement}$	$::=$	$\mathit{Binding} \mid \mathit{Command}$
$\mathit{Binding}$	$::=$	$\mathit{MV}_m = E_m$
$\mathit{Command}$	$::=$	$\mathsf{P\_store}\ (\mathit{SE},\ \mathit{SE},\ \mathit{SE}) \mid \mathsf{Cons\_store\_1}\ (\mathit{SE},\ \mathit{SE}) \mid \mathsf{Cons\_store\_2}\ (\mathit{SE},\ \mathit{SE}) \mid \mathsf{Store\_error} \mid \top_s$
$\mathit{Program}$	$::=$	Block_1

Figure 1: The Grammar of Kid

2 Kid : The Kernel Id Language

Id is a high level functional language [13] augmented with a novel data structuring facility known as I-structures [3]. Id, like most modern functional languages, is non-strict, and has higher-order functions and a Milner-style type system. It also has a fairly large syntax to express currying, loops, list and array comprehensions, and pattern matching for all algebraic types. We have found it difficult to give direct operational semantics of Id because of its syntactic complexities. Kid, a kernel language for Id, has been developed to give precise (though indirect) operational semantics of Id. Kid has also proved to be an extremely useful intermediate language for the Id compiler [2]. Within the compiler, all machine independent optimizations are expressed as source-to-source program transformations on Kid. We believe I-structures are an essential feature of Kid, and are needed to express the semantics of the purely functional subset of Id.

Kid has only uncurried operators and no complex expressions. A major subset of Kid is simply the λ-calculus with constants and let-blocks. However, unlike other functional languages, let-blocks play a fundamental role, that is, they can not be eliminated by systematic substitution of variables. The syntax of Kid is given in Figure 1. Every expression, except a block or λ-expression, consists of a function symbol followed by the corresponding number of arguments. *The ordering of statements within a block is irrelevant.*

An important feature of Kid is the concept of *multiple values*. The expression

$$x,\ y = \{_2\ a = \cdots;\ b = \cdots;\ \mathsf{In}\ a,\ b\}$$

is a well-formed-expression, where "x, y" indicates multiple variables, not to be confused with a 2-tuple in Id [13]. The 2 after the curly brace indicates that two values are to be returned by this block expression. Multiple values avoid packaging values in a data structure, and are useful in expressing some optimizations. Thus, in Kid a binding has the form $MV = E$, where MV stands for multiple variables. Suppose we have m variables on the left-hand-side, then the expression E on the right-hand-side must return m values. In the sequel we capture the number of values that an expression produces by subscripting the corresponding syntactic category. Thus, to express the above binding we will write $MV_m = E_m$. Note that the function symbol Apply appears as a $PF2_1$ in the grammar because we assume that all user-defined procedures return only one result. We also use subscripted function symbols to express a family of functions. For example, $\mathsf{Make\_tuple}_n$ stands for $\mathsf{Make\_tuple}_2$, $\mathsf{Make\_tuple}_3$, *etc.* . Subscripts in a function symbol do not necessarily represent the number of values to be returned by the application of the function. By convention, we drop the subscript when its value is one.

Kid also contains While-loops and For-loops because these play a significant role in program transformations. These constructs also play a major role when Kid is further translated into a still lower-level language and machine code [2].

We briefly describe the use of I-structures in the translation of list comprehensions. A typical translation of a list-comprehension is given in terms of nested map-list operations followed by a list flattening operation [15]. In Id, we make use of "open lists" , a type of I-structure, to generate a tail recursive program. The translation of the list comprehension expression `{: e || x <- xs ; y <- ys}` may be fiven as follows:

```
{ h1  =  Open_cons ();
  hn  = { For x <- xs do
             Next h1 =
               { Case ys of
                  | Nil = h1
                  | y:yss =
                      { h =  Open_cons ();
                        h1.cons_2 = h.cons_2;
                       In { For y <- ys do
                              t  = Open_cons ();
                              t.cons_1 = e;
                              h.cons_2 = t;
                              Next h = t;
                           Finally h }}};
          Finally h1 };
  hn.cons_2 = Nil;
 In h1.cons_2 }
```

Basically, in the above program, a open list (signified by h in the inner loop) is generated for each element of xs and then these open lists are "glued" together in the outer loop.

A translation of functional subset of Id including I-structures is given in [2].

We now digress to introduce contextual reduction systems before giving the calculus and the operational semantics of Kid.

3 Contextual Reduction Systems

A Contextual Reduction System is a pair (A(F, S), R) , where A is a set of terms, and R is a set of context sensitive rules. F includes at least the Apn,m primitive. The saliant syntactic features of the terms in A can be seen by the grammar given in Figure 2. This grammar can be related to the Kid grammar by defining F to be +, Apply, WLoop_m, P_select, *etc.* and S to be P_store, Cons_store, *etc.* respectively.

$$\begin{array}{lll}
SE & \in & \text{Simple Expression} \\
E_m & \in & \text{Expression with } m \text{ outputs} \\
\mathsf{Fn,m} & \in & \text{Primitive Function of } n \text{ arguments and } m \text{ outputs} \\
\mathsf{Sn} & \in & \text{Primitive command of } n \text{ arguments}
\end{array}$$

$$\begin{array}{lll}
SE & ::= & Variable \mid \mathsf{F}_0 \\
SE_m & ::= & \underbrace{SE, \cdots, SE}_{m} \\
MV_m & ::= & \underbrace{Variable, \cdots, Variable}_{m} \\
lambda_E & ::= & \lambda\mathsf{n,m}\ (MV_n)\,.\,(E_m) \\
E_m & ::= & SE_m \mid \mathsf{Fn,m}\ (SE_n) \mid lambda_E \mid Block_m \\
Block_m & ::= & \{\ [Statement;]^* \ \mathsf{In}\ SE_m\} \\
Statement & ::= & Binding \mid Command \\
Binding & ::= & MV_m = E_m \\
Command & ::= & \mathsf{S}_n\ (SE_n) \\
Term & ::= & Block
\end{array}$$

Figure 2: Syntax of terms of a CRS

3.1 Basic Definitions

The applicability of a context sensitive rule depends upon both the structure of the term and a precondition on the context in which the term occurs. For example, the context sensitive rule corresponding to the well known rule " $\mathsf{Cons\_1}\ (\mathsf{Cons}\ X_1\ X_2) \longrightarrow X_1$" is

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\mathsf{Cons\_1}\ (X) \quad \longrightarrow \quad X_1}$$

where the binding $X = \mathsf{Cons}\ (X_1,\ X_2)$ over the line denotes a precondition, and X, X_1, and X_2 denote meta-variables. A way of reading the above rule is that $\mathsf{Cons\_1}\ (X)$ can be rewritten to X_1 if the binding $X = \mathsf{Cons}\ (X_1,\ X_2)$ occurs in the context of $\mathsf{Cons\_1}\ (X)$.

A *context* is a term with an arbitrary number of $\Box_E$, expression holes, and $\Box_S$, statement holes. Formally contexts are terms generated by the grammar of Figure 2, where $\Box_E$ and $\Box_S$ are added to SE and *Statement* categories, respectively. Thus, given a context $C[\Box_S,\ \Box_E]$, and a *substitution*, σ, for the meta-variables X, X_1 and X_2 such that

$$C[(X = \mathsf{Cons}\ (X_1,\ X_2))^\sigma,\ (\mathsf{Cons\_1}\ (X))^\sigma]$$

is a term, we say

$$C[(X = \mathsf{Cons}\ (X_1,\ X_2))^\sigma,\ (\mathsf{Cons\_1}\ (X))^\sigma] \longrightarrow C[(X = \mathsf{Cons}\ (X_1,\ X_2))^\sigma,\ X_1^\sigma] \tag{1}$$

where $\longrightarrow$ stands for "rewrites to". From (1) we can observe that the precondition of the rule is not affected by the rewriting, however, the precondition usually affects the outcome of the rewriting.

Definition 3.1 (Context sensitive rule) *A rule is an ordered set of preconditions, $P_1 \cdots P_n$, and a left-hand-side, l, and a right-hand-side, r, and is written as*

$$\frac{P_1 \mid \cdots \mid P_n}{l \;\longrightarrow\; r}$$

where P_i is a statement and l and r are expressions.

Definition 3.2 (Redex) *Given a rule $\frac{P_1 \mid \cdots \mid P_n}{l \;\longrightarrow\; r}$ and a term $M \equiv C[S_1, \cdots, S_n, E]$, E is said to be a* **redex** *iff*

(1) $E \equiv l^\sigma$;

(2) $S_j \equiv P_j^\sigma \quad \forall\, j,\, 1 \le j \le n$;

where σ is a substitution.

Definition 3.3 (Reduction Relations) *Given terms M, $N \in A$ and a rule $\tau \in R$, M reduces to N in one step ($M \underset{\tau}{\longrightarrow} N$), iff $M \equiv C[P_1^\sigma, \cdots, P_n^\sigma, l^\sigma]$, and $N \equiv C[P_1^\sigma, \cdots, P_n^\sigma, r^\sigma]$.*
The one-step rewriting relation $\underset{R}{\longrightarrow} = \bigcup_{\tau \in R}(\underset{\tau}{\longrightarrow})$.
The transitive reflexive closure of $\underset{R}{\longrightarrow}$ is written as $\underset{R}{\twoheadrightarrow}$.

3.2 Basic Rules of Contextual Reduction System's ($R_{\mathbf{CRS}}$)

One has to deal with the problem of free-variable capture in any formal system with bound variables. In the λ-calculus, the problem is usually solved either by making some *variable convention*, as for example, assuming that all free variables are different from the bound variables, or by adopting a *variable-free notation* such as that of DeBruijn [8]. According to the variable convention given in [4], the term $(\lambda x.xx)(\lambda x.xx)$ is a legal term, while the term $(\lambda y.(\lambda x.+(x,y))x$ is not a legal term, because the variable x appears both free and bound. The convention allows one to express application in terms of a *naive* form of substitution, which does not require α-renaming at execution time. In a system with let-blocks, in order to avoid the free-variable capture as well as a to allow naive substitution, we have to adopt a even more stringent convention: *all* bound variables in an expression must be unique and different from variables free in the whole program. In order to maintain this invariant we will occasionally rename all bound variables of a subterm to completely new variables explicitly, by applying the function $\mathcal{RB}$ to the subterm. For example,

$$\mathcal{RB}\,[\![\{x = +\,(a,\ 1)\ \mathsf{In}\ x\}]\!] = \{x' = +\,(a,\ 1)\ \mathsf{In}\ x'\}$$

An example of the use of $\mathcal{RB}$ function arises in the use of the application rule (see Section 4.1) as shown below

$$\frac{F = \lambda_{\mathsf{n,m}}\,(\vec{Z_n})\,.\,(E)}{\mathsf{Ap}_{\mathsf{n,m}}\,(F,\ \vec{X_n}) \;\longrightarrow\; (\mathcal{RB}[\![E]\!])\,[\vec{X_n}\ /\ \vec{Z_n}]}$$

All contextual reduction systems have the following rules, named R_{CRS}.

Substitution rules

$$\frac{X = Y}{X \longrightarrow Y}$$

$$\frac{X = C}{X \longrightarrow C}$$

where meta-variable C stands for a constant, and X and Y stand for distinct variables.

Block Flattening rule

$$\begin{array}{lcl} \{\mathsf{m}\ \vec{X_n} = \{\mathsf{n}\ SS_1;\ SS_2;\ \cdots & & \{\mathsf{m}\ \vec{X_n} = \vec{Y_n}; \\ \qquad\qquad\qquad \mathsf{In}\ \vec{Y_n}\} & \longrightarrow & \quad SS_1;\ SS_2;\ \cdots \\ \quad S_1;\ \cdots\ S_n & & \quad S_1;\ \cdots\ S_n \\ \quad \mathsf{In}\ \vec{Z_m}\} & & \quad \mathsf{In}\ \vec{Z_m}\} \end{array}$$

Multivariable rule

$$\vec{X_n} = \vec{Y_n} \longrightarrow (X_1 = Y_1; \cdots X_n = Y_n)$$

Lemma 3.4 *R_{CRS} is Strongly Normalizing (SN) and Church-Rosser (CR).*

Proof: Since there are only a finite number of blocks and occurrences of a variable in a term M, R_{CRS} is strongly normalizing. It is easy to see that the block flattening rules are *locally confluent*. Since variable substitution rules within a flattened block are also *locally confluent*, the uniqueness of the normal form of M follows trivially from Newman's Lemma [4]. ∎

3.3 Canonical Forms of terms in a CRS

The notion of α-equivalence between terms as defined in the λ-calculus, makes too fine a distinction in a CRS calculus, for example, the following two Kid terms have apparently different syntactic structure.

$$\{x = 8;\ t' = \{y = x;\ t = x + y \ \mathsf{In}\ t\}\ \mathsf{In}\ t'\} \quad \text{and} \quad \{x = 8;\ y = x;\ t = x + y\ \mathsf{In}\ t\}$$

However, we consider the difference between the above two terms merely "syntactic noise". Eliminating this syntactic noise is the motivation behind the following definitions.

Definition 3.5 (Canonical form) *Let N be the normal form of a term M with respect to R_{CRS}. $\overline{M}$, the canonical form of M, is the term obtained by deleting from N all bindings of the form $x = y$ (where x and y are distinct variables) or $x = v$ (where v is a constant).*

Definition 3.6 (α-equivalence) *Two terms M and N are said to be α-equivalent, if $\overline{M}$ and $\overline{N}$ can be transformed into each other by a consistent renaming of bound variables.*

Lemma 3.7 *Each term has a unique canonical form.*

4 Contextual Reduction System of Kid

We now present a set of rewrite rules, R_{Kid}, which give an intuitive understanding of how Kid terms may get evaluated. If we call A the set of Kid terms generated by the grammar of Figure 1, then the structure $\langle A, R_{Kid}\rangle$ is a Contextual Reduction System. We assume that a primitive function is applied only to arguments of appropriate types, i.e., the type checking has been done statically.

All the variables that appear on the left-hand-side of the rules are meta-variables that range over appropriate syntactic categories. By convention, we use capital letters for meta-variables and small letters for Kid variables. All variables that appear on the right-hand-side of the rules are either meta-variables or "new" Kid variables. We will make use of the following convention regarding meta-variables:

$$\begin{array}{lcl} X_i,\ Z_i,\ Y_i,\ F_i,\ P,\ B,\ U,\ D & \in & \text{Variable and Constant} \\ C & \in & \text{Constant} \\ S_i,\ SS_i,\ S' & \in & \text{Statement} \\ E_i & \in & \text{Expression} \end{array}$$

The notation $E\,[Y/X]$ means the substitution of Y for X in E. Due to our assumptions $E\,[Y/X]$ will simply indicate naive substitution, that is, substitution where no danger of free-variable-capture exists and where X can be replaced by Y without regards to scope. Moreover, we will use the notation $\overrightarrow{X_{n,m}}$ to stand for $(X_n, \cdots, X_m)$, $\overrightarrow{X_m}$ for $\overrightarrow{X_{1,m}}$, and $E\,[\overrightarrow{Y_n}\ /\ \overrightarrow{X_n}]$ for $E\,[Y_1/X_1, \cdots, Y_n/X_n]$, which is the same as $(\cdots((E\,[Y_1/X_1])\,[Y_2/X_2])\cdots)\,[Y_n/X_n])$.

In the following, $\underline{n}$ represents a numeral.

4.1 Kid Rewrite Rules

δ rules

$$\begin{array}{lcll} +\,(\underline{m},\ \underline{n}) & \xrightarrow[\delta]{} & \underline{+(m,\ n)} & \\ & \vdots & & \\ \text{Equal?}\ (\underline{n},\ \underline{n}) & \xrightarrow[\delta]{} & \text{True} & \\ \text{Equal?}\ (\underline{m},\ \underline{n}) & \xrightarrow[\delta]{} & \text{False} & \qquad (\text{if m} \neq n) \end{array}$$

Case rules

$$\begin{array}{lcl}
\text{Bool_casem (True, } E_1, E_2) & \longrightarrow & E_1 \\
\text{Bool_casem (False, } E_1, E_2) & \longrightarrow & E_2 \\
\text{List_casem (Nil, } E_1, E_2) & \longrightarrow & E_1
\end{array}$$

$$\frac{X = \text{Open\_cons ()}}{\text{List\_casem } (X, E_1, E_2) \longrightarrow E_2}$$

Arity Detection rule

$$\frac{F = \lambda(Z).E}{\text{Apply } (F, X) \longrightarrow \text{Ap } (F, X)}$$

$$\frac{F = \lambda_n(\vec{Z_n}).E}{\text{Apply } (F, X) \longrightarrow \text{Apply}_1 (F, \underline{n}, X)} \quad n > 1$$

Similar rules apply for $\underline{\lambda_n}$.

$$\frac{F_i = \text{Apply}_i (F, \underline{n}, \vec{X_i})}{\text{Apply } (F_i, X) \longrightarrow \text{Apply}_{i+1} (F, \underline{n}, \vec{X_{i+1}})} \quad i < (n-1)$$

$$\frac{F_i = \text{Apply}_i (F, \underline{n}, \vec{X_i})}{\text{Apply } (F_i, X) \longrightarrow \text{Apn } (F, \vec{X_n})} \quad i = (n-1)$$

Application rule

$$\frac{F = \lambda\text{n,m } (\vec{Z_n}) \, . \, (E)}{\text{Apn,m } (F, \vec{X_n}) \longrightarrow (\mathcal{RB}[\![E]\!]) \, [\vec{X_n} \, / \, \vec{Z_n}]}$$

A similar rule applies for $\underline{\lambda_{\text{n,m}}}$.

Loop rules

$$\begin{array}{lcl}
\text{WLoopn } (P, B, \vec{X_n}, \text{True}) & \longrightarrow & \{\text{n } \vec{t_n} \; = \; \text{Apn,n } (B, \vec{X_n}); \\
 & & \quad t_p \; = \; \text{Apn } (P, \vec{t_n}); \\
 & & \quad \vec{t'_n} \; = \; \text{WLoopn } (P, B, \vec{t_n}, t_p) \\
 & & \quad \text{In } \vec{t'_n}\} \\
\text{WLoopn } (P, B, \vec{X_n}, \text{False}) & \longrightarrow & \vec{X_n}
\end{array}$$

In the following two rules we assume that the index variable is the first variable in $\vec{X_n}$.

$$
\begin{array}{lcll}
\mathsf{FLoopn}\ (U,\ D,\ B,\ \vec{X_n},\ \mathsf{True}) & \longrightarrow & \{\mathsf{n}\ \vec{t_{2,n}} & = \ \mathsf{Ap}_{\mathsf{n,n-1}}\ (B,\ \vec{X_n}); \\
 & & \quad t_1 & = \ +\ (X_1,\ D); \\
 & & \quad t_p & = \ <(t_1,\ U); \\
 & & \quad \vec{t'_n} & = \ \mathsf{FLoopn}\ (U,\ D,\ B,\ \vec{t_n},\ t_p) \\
 & & \quad\quad \mathsf{In}\ \vec{t'_n}\} & \\
\mathsf{FLoopn}\ (U,\ D,\ B,\ \vec{X_n},\ \mathsf{False}) & \longrightarrow & \vec{X_n} &
\end{array}
$$

Tuple rule

$$
\frac{X = \mathsf{Make\_tuplen}\ (\vec{X_n})}{\mathsf{Detuplen}\ (X) \ \longrightarrow\ \vec{X_n}}
$$

List rules

$$
\begin{array}{lcl}
\mathsf{Cons}\ (X,\ Y) & \longrightarrow & \{\ t \ = \ \mathsf{Open\_cons}\ (); \\
 & & \quad \mathsf{Cons\_store\_1}\ (t,\ X); \\
 & & \quad \mathsf{Cons\_store\_2}\ (t,\ Y) \\
 & & \quad\quad \mathsf{In}\ t\}
\end{array}
$$

$$
\frac{\mathsf{Cons\_store\_1}\ (X,\ Y)}{\mathsf{Cons\_1}\ (X) \ \longrightarrow\ Y}
$$

$$
\frac{\mathsf{Cons\_store\_2}\ (X,\ Y)}{\mathsf{Cons\_2}\ (X) \ \longrightarrow\ Y}
$$

$$
\frac{\mathsf{Cons\_store\_1}\ (X,\ Y)}{\mathsf{Cons\_store\_1}\ (X,\ Y') \ \longrightarrow\ \top_s}
$$

$$
\frac{\mathsf{Cons\_store\_2}\ (X,\ Y)}{\mathsf{Cons\_store\_2}\ (X,\ Y') \ \longrightarrow\ \top_s}
$$

Array rules

$$
\frac{X = \mathsf{I\_array}\ (X_b)}{\mathsf{Bounds}\ (X) \ \longrightarrow\ X_b}
$$

$$
\frac{\mathsf{P\_store}\ (X,\ Y,\ Z)}{\mathsf{P\_select}\ (X,\ Y) \ \longrightarrow\ Z}
$$

$$
\frac{\mathsf{P\_store}\ (X,\ Y,\ Z)}{\mathsf{P\_store}\ (X,\ Y,\ Z') \ \longrightarrow\ \top_s}
$$

Propagation of $\top$

$$\{\mathsf{m}\ X = \top;\ S_1; \cdots S_n\ \mathsf{In}\ \vec{Z_m}\ \} \longrightarrow \top$$
$$\{\mathsf{m}\ \top_s;\ S_1; \cdots S_n\ \mathsf{In}\ \vec{Z_m}\ \} \longrightarrow \top$$

The rules for propagating $\top$ were motivated by a discussion with Vinod Kathail.

Theorem 4.1 *Kid is Confluent upto α-renaming on canonical terms.*

Proof: See [1]. ∎

4.2 Printable Values and Answer of a Kid Term

We now define the printable information associated with a term. The grammar for printable values for Kid is given in Figure 3. A precise notion of printable values is essential to develop an interpreter for Kid as well as to discuss the correctness of optimizations (see Sections 4.3 and 5.3, respectively).

Atoms	::=	*Integers* | *Booleans* | Error | "Function" | Ω
List	::=	$\langle$List *PV* *List*$\rangle$ | Nil
Tuple	::=	$\langle$2_Tuple *PV* *PV*$\rangle$ | $\langle$3_Tuple *PV* *PV* *PV*$\rangle$ | $\cdots$
Array	::=	$\langle$2_Array *Tuple* *PV* *PV*$\rangle$ |$\rangle$3_Array *Tuple* *PV* *PV* *PV*$\rangle$ | $\cdots$
PV	::=	*Atoms* | *List* | *Tuple* | *Array* | $\top$

Figure 3: Grammar of Printable Values

The following procedure, $\mathcal{P}$, produces the printable value associated with a term. ρ and σ represent, respectively, the list of bindings that have as RHS either a λ-expression or an allocator, *i.e.* Make_tuple, I_array, Open_cons, and the list of store commands, *i.e.* the I-structure store. The procedure $\mathcal{L}$ is used to lookup the value of a variable or a location in ρ and σ, respectively. Given a program, *i.e.* a closed term, M, the $\mathcal{P}rint$ procedure is invoked as follows:

$$\mathcal{P}rint\ (M) = \mathcal{P}\ (\overline{M},\ \mathsf{Nil},\ \mathsf{Nil}) \quad \text{with } \overline{M} \text{ the canonical form of } M.$$

where $\mathcal{P}$ is:

$\mathcal{P}[\![\{S_s;\ A_s;\ B_s\ \mathsf{In}\ X\}]\!]\ \rho\ \sigma = \mathcal{P}[\![X]\!]\ (A_s : \rho)\ (S_s : \sigma)$

$\mathcal{P}[\![\underline{n}]\!]\ \rho\ \sigma = \underline{n}$

$\mathcal{P}[\![\mathsf{True}]\!]\ \rho\ \sigma = \mathsf{True}$

$\mathcal{P}[\![\mathsf{False}]\!]\ \rho\ \sigma = \mathsf{False}$

$\mathcal{P}[\![\top]\!]\ \rho\ \sigma = \top$

$\mathcal{P}[\![\mathsf{Nil}]\!]\ \rho\ \sigma = \mathsf{Nil}$

$$
\mathcal{P}[\![X]\!]\,\rho\,\sigma = \begin{cases} \langle \textsf{n\_Tuple}\ (\mathcal{P}[\![X_1]\!]\,\rho\,\sigma)\cdots(\mathcal{P}[\![X_n]\!]\,\rho\,\sigma)]\!]\rangle & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \textsf{Make\_tuplen}(X_1,\cdots,X_n) \\ \langle \textsf{List}\ (\mathcal{P}[\![X.1]\!]\,\rho\,\sigma)\ (\mathcal{P}[\![X.2]\!]\,\rho\,\sigma)]\!]\rangle & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \textsf{Open\_cons}() \\ \langle \textsf{Array}\ \langle \textsf{2\_Tuple}\ \underline{l}\ \underline{u}\rangle\ (\mathcal{P}[\![X.\underline{l}]\!]\,\rho\,\sigma)\cdots(\mathcal{P}[\![X.\underline{u}]\!]\,\rho\,\sigma)]\!]\rangle & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \textsf{I\_array}(Y) \\ & \text{and } \mathcal{L}(\mathrm{Y},\,\rho) = \textsf{Make\_tuple}\ (\underline{l},\ \underline{u}) \\ \text{“Function”} & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \lambda\ \vec{X_n}\ .\ (E)\ \text{ or} \\ & \text{if } \mathcal{L}(\mathrm{X},\,\rho) = \textsf{Apply}_{\textsf{i}}\ (F,\ \underline{n},\ \vec{X_i})\ \wedge\ i < (n-1) \\ \Omega & \text{Otherwise} \end{cases}
$$

$$
\mathcal{P}[\![X.i]\!]\,\rho\,\sigma = \begin{cases} \mathcal{P}[\![Y]\!]\,\rho\,\sigma & \text{if } \mathcal{L}(\mathrm{X.i},\,\sigma) = Y \\ \Omega & \text{if } \mathcal{L}(\mathrm{X.i},\,\sigma) = \text{not found} \end{cases}
$$

Notice that the printable value of a finite term can be an infinite tree. For example, the $\mathcal{P}$rint of :

```
{x = Open_cons ();
 Cons_1 (x, 1);
 Cons_2 (x, x);
  In x}
```

is $\langle$ List 1 $\langle$ List 1 $\langle\cdots\rangle\cdots\rangle$ $\rangle$, that is, an infinite list of 1's.

We want to define the answer associated with a term in terms of printable values. Furthermore, we want to describe the answer independently of any interpreter, *i.e.* method of computing it. We need to define an ordering on printable values.

Definition 4.2 (Partial Order on PV) *Let a and b be Printable Values; $a \sqsubseteq b$ iff*

(i) $a = \Omega$*; or*

(ii) $b = \top$*; or*

(iii) both a and b are Atoms and $a = b$; or

(iv) $a = \langle \textsf{n\_Tuple}\ a_1 \cdots a_n\rangle$ *and* $b = \langle \textsf{n\_Tuple}\ b_1 \cdots b_n\rangle$ *and* $a_i \sqsubseteq b_i\ \forall\ 1 \le i \le n$*; or*

(v) $a = \langle \textsf{List}\ a_1\ a_2\rangle$ *and* $b = \langle \textsf{List}\ b_1\ b_2\rangle$ *and* $a_1 \sqsubseteq b_1$ *and* $a_2 \sqsubseteq b_2$*; or*

(vi) $a = \langle \textsf{n\_Array}\ abounds\ a_1 \cdots a_n\rangle$ *and* $b = \langle \textsf{n\_Tuple}\ bbounds\ b_1 \cdots b_n\rangle$ *and* $abounds = bbounds$ *and* $a_i \sqsubseteq b_i\ \forall\ 1 \le i \le n$.

Theorem 4.3 *PV is a complete partial order with respect to* $\sqsubseteq$.

For the proof see [7].

Lemma 4.4 *Given a term M,* $M \longrightarrow N \implies \mathcal{P}rint\ (M) \sqsubseteq \mathcal{P}rint\ (N)$.

Pictorially:

$$
\begin{array}{ccccccc} M & \longrightarrow & M_1 & \longrightarrow & M_2 & & \cdots \\ \mathcal{P}rint(M) & \sqsubseteq & \mathcal{P}rint(M_1) & \sqsubseteq & \mathcal{P}rint(M_2) & & \cdots \end{array}
$$

Intuitevely, the answer of a term may be defined as the limit of this chain, that is, the answer is the maximum printable value that can be obtained by reducing a term. We need the following definitions to give a precise definition of "Answer".

Definition 4.5 (Reduction Graph of a term) *Given a term M, the reduction graph of M, $\mathcal{G}(M)$, is defined as*

$$\{N \mid M \twoheadrightarrow N\}.$$

Definition 4.6 (Printable Reduction Graph of a term) *Given a term M, the printable reduction graph of M, $\mathcal{PG}(M)$, is defined as*

$$\{\mathcal{P}rint\ (N) \mid M \twoheadrightarrow N\}.$$

Definition 4.7 (Answer) *Given a term M, the Answer of M, $\mathcal{P}rint^*(M)$, is defined as*

$$\sqcup\, \mathcal{PG}(M).$$

In order for the above definition to make sense we need to show that the limit of the printable reduction graph exists and is unique.

Theorem 4.8 $\forall$ *M, $\mathcal{PG}(M)$ is directed.*

Proof: Given a, $b \in \mathcal{PG}$(M), we have:

$$\exists M_1, M_2 \text{ such that } a = \mathcal{P}rint(M_1),\ M \twoheadrightarrow M_1 \ \wedge\ b = \mathcal{P}rint(M_2),\ M \twoheadrightarrow M_2.$$

By confluence, $\exists$ M_3 such that

$$M_1 \twoheadrightarrow M_3 \ \wedge\ M_2 \twoheadrightarrow M_3$$

By lemma 4.4,

$$a \sqsubseteq c \ \wedge\ b \sqsubseteq c,\ c = \mathcal{P}rint(M_3)$$

■

Since $\mathcal{PG}$(M) $\subseteq PV$ is directed, by theorem 4.3 we know that the limit exists and is unique.

4.3 An Interpreter to Compute the Answer of a Kid term

The set of rewrite rules given in Section 4 per se do not define the operational semantics of Kid, they define a calculus to prove properties of programs. From a computational point of view we need to specify a *strategy* to apply these rules. Furthermore, the reduction strategy should be normalizing with respect to the definition of the answer. Intuitevely the interpreter should stop when it is known that no more printable information can be gotten by further reductions.

The strategy consists in evaluating the outermost redexes first, and in case of a block, evaluating all the right-hand-side redexes in parallel. We will represent a block as $\{Ss;\ As;\ Bs \text{ In } X\}$, where

- Ss represents a set of store commands in the block
- As represents a set of λ-bindings and allocator bindings in the block
- Bs represents a set of bindings, excluding λ-binding and allocators, in the block

The interpreter $\mathcal{E}$ keeps track of store commands it encounters in σ, an I-structure store. It also keeps track of λ-bindings and allocator bindings encountered in ρ. At the start of the evaluation, both ρ and σ are considered to be empty.

The rules for propagating $\top$ imply that whenever new store commands are added to the store a check has to be made for inconsistencies, that is, multiple writes into the same location. We will assume this function is performed by $\mathcal{CC}$. Thus, $\mathcal{CC}\ (\rho \cup Ss)$ returns either a new consistent store or $\top_s$. While evaluating a block, we need to substitute variables and constants and eliminate the corresponding binding. We also need to flatten blocks. Though all nested blocks can be flattened in one go, we need to merge in one step only the block expressions on the RHS of bindings in the top-level block. Any substitution or block flattening can create new redexes on the RHS of bindings in a block. According to the $\top$ propagation rules, if a binding of the form $\mathbf{x} = \top$ exists in any block then that block goes to $\top$. This can also be done during the substitution process. We will assume a function $\mathcal{F}\&\mathcal{S}$ (for flatten and substitute) that accomplishes all this. In addition to the new block, it returns a flag showing if any substitutions or flattening was done. Thus,

$$\begin{aligned}\mathcal{F}\&\mathcal{S}[\![\{X = 1; Y = \{\mathsf{P\_store}\ (W,\ I,\ Z);\ Z = X + 3 \text{ In } Z\} \text{ In } Y\}]\!]\\ = \{\mathsf{P\_store}\ (W,\ I,\ Z);\ Z = 1 + 3;\ \text{In } Z\}, \mathsf{True}\end{aligned}$$

We will also assume the existence of a function $\mathcal{ERHS}$ which is only applied to a block expression. It may be described informally as

$$\begin{aligned}\mathcal{ERHS}[\![\{Ss;\ As;\ X_1 = E_1; \cdots X_n = E_n \text{ In } X_i\}]\!]\ \rho\ \sigma =\\ \{Ss;\ As;\ X_1 = \mathcal{E}[\![E_1]\!]\ \rho\ \sigma; \cdots X_n = \mathcal{E}[\![E_n]\!]\ \rho\ \sigma \text{ In } X_i\}\end{aligned}$$

Given a program M, the $\mathcal{E}$ procedure is invoked as follows:

$$\mathcal{E}val\ [\![M]\!] = \mathcal{E}[\![M]\!]\ \mathsf{Nil}\ \mathsf{Nil}$$

The Interpreter:

$\mathcal{E}[\![+ (\underline{m},\ \underline{n})]\!]\ \rho\ \sigma = \underline{+(m,\ n)}$

$\mathcal{E}[\![\mathsf{Equal?}\ (\underline{n},\ \underline{n})]\!]\ \rho\ \sigma = \mathsf{True}$

$\mathcal{E}[\![\mathsf{Equal?}\ (\underline{n},\ \underline{m})]\!]\ \rho\ \sigma = \mathsf{False}$

$\mathcal{E}[\![\mathsf{Bool\_casem}\ (\mathsf{True},\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{E}[\![E_1]\!]\ \rho\ \sigma$

$\mathcal{E}[\![\mathsf{Bool\_casem}\ (\mathsf{False},\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{E}[\![E_2]\!]\ \rho\ \sigma$

$\mathcal{E}[\![\mathsf{List\_casem}\ (\mathsf{Nil},\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{E}[\![E_1]\!]\ \rho\ \sigma$

$\mathcal{E}[\![\mathsf{List\_casem}\ (X,\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{E}[\![E_2]\!]\ \rho\ \sigma \qquad \text{if } (\mathcal{L}\ (X,\ \rho)) = \mathsf{Open\_cons}\ ()$

$$\mathcal{E}[\![\mathsf{Apply}\ (F,\ X)]\!]\ \rho\ \sigma = \begin{cases} \mathcal{E}[\![\mathsf{Ap}\ (F,\ X)]\!]\ \rho\ \sigma & \text{if } (\mathcal{L}\ (F,\ \rho)) = \lambda Z\ .\ (E) \\ \mathsf{Apply_1}\ (F,\ \underline{n},\ X) & \text{if } (\mathcal{L}\ (F,\ \rho)) = \lambda_n(\vec{Z_n}).(E)\ \wedge\ n > 1 \\ \mathsf{Apply_{i+1}}\ (F,\ \underline{n},\ \vec{X_{i+1}}) & \text{if } (\mathcal{L}\ (F,\ \rho)) = \mathsf{Apply_i}\ (F,\ \underline{n},\ \vec{X_i})\ \wedge\ i < (n-1) \\ \mathcal{E}[\![\mathsf{Apn}\ (F,\ \vec{X_n})]\!]\ \rho\ \sigma & \text{if } (\mathcal{L}\ (F,\ \rho)) = \mathsf{Apply_i}\ (F,\ \underline{n},\ \vec{X_i})\ \wedge\ i = (n-1) \end{cases}$$

$\mathcal{E}[\![\mathsf{Apn,m}\ (F,\ \vec{X_n})]\!]\ \rho\ \sigma = \mathcal{E}[\![(\mathcal{RB}\ [\![E]\!])\ [\vec{X_n}\ /\ \vec{Z_n}]]\!]\ \rho\ \sigma$ where $(\mathcal{L}\ (F,\ \rho)) = \lambda_{n,m}(\vec{Z_n}).E$

$$\begin{aligned} \mathcal{E}[\![\mathsf{WLoopn}\ (P,\ B,\ \vec{X_n},\ \mathsf{True})]\!]\ \rho\ \sigma = \mathcal{E}[\![\{\mathsf{n}\ \vec{t_n} &= \mathsf{Apn,n}\ (B,\ \vec{X_n}); \\ t_p &= \mathsf{Apn}\ (P,\ \vec{t_n}); \\ \vec{t'_n} &= \mathsf{WLoopn}\ (P,\ B,\ \vec{t_n},\ t_p) \\ &\mathsf{In}\ \vec{t'_n}\}]\!]\ \rho\ \sigma \end{aligned}$$

$\mathcal{E}[\![\mathsf{WLoopn}\ (P,\ B,\ \vec{X_n},\ \mathsf{False})]\!]\ \rho\ \sigma = \vec{X_n}$

$\mathcal{E}[\![\mathsf{Detuple_n}\ (X)]\!]\ \rho\ \sigma = \vec{X_n}$ if $(\mathcal{L}\ (X,\ \rho)) = \mathsf{Make\_tuple}\ \vec{X_n}$

$\mathcal{E}[\![\mathsf{Cons}\ (X,\ Y)]\!]\ \rho\ \sigma = \mathcal{E}[\![\{t = \mathsf{Open\_cons}\ ();\ \mathsf{Cons\_store\_1}\ (t,\ X);\ \mathsf{Cons\_store\_2}\ (t,\ Y)\ \mathsf{In}\ t\}]\!]\ \rho\ \sigma$

$\mathcal{E}[\![\mathsf{Cons\_1}\ (X)]\!]\ \rho\ \sigma = Y$ if $(\mathcal{L}\ (X,\ \sigma)) = \mathsf{Cons\_store\_1}\ (X,\ Y)$

$\mathcal{E}[\![\mathsf{Cons\_2}\ (X)]\!]\ \rho\ \sigma = Y$ if $(\mathcal{L}\ (X,\ \sigma)) = \mathsf{Cons\_store\_2}\ (X,\ Y)$

$\mathcal{E}[\![\mathsf{Bounds}\ (X)]\!]\ \rho\ \sigma = X_b$ if $(\mathcal{L}\ (X,\ \rho)) = \mathsf{I\_array}\ (X_b)$

$\mathcal{E}[\![\mathsf{P\_select}\ (X,\ Y)]\!]\ \rho\ \sigma = Z$ if $(\mathcal{L}\ (X,\ \sigma)) = \mathsf{P\_store}\ (X,\ Y,\ Z)$

$\mathcal{E}[\![\{Ss;\ As;\ Bs\ \mathsf{In}\ X\}]\!]\ \rho\ \sigma = e$

where e is obtained by the execution of the following program written in pseudo-Id

```
{ blk, - = F&S({Ss; As; Bs In X});
  flag = True;
  In If blk = ⊤ then ⊤
             else {While flag do
                     Suppose blk is {Ss; As; Bs In X}
                   is = CC(σ ∪ Ss);
                   fs = ρ ∪ As;
                   next blk, next flag =  If is = ⊤s then (⊤, False)
                                          else F&S(E_RHS(blk, is, fs))
                   finally blk}}
```

If none of the above clauses apply, then

$\mathcal{E}[\![E]\!]\ \rho\ \sigma = E$

Theorem 4.9 (Soundness) *Given a term M,*

$$\mathcal{E}val[\![M]\!] = N \Longrightarrow \mathcal{P}rint^*(M) = \mathcal{P}rint(N)$$

Theorem 4.10 (Normalisation Theorem) *Given a term M,*

$$\mathcal{P}rint^*(M)\ \textit{is finite} \Longrightarrow \mathcal{P}rint(\mathcal{E}[\![M]\!]) = \mathcal{P}rint^*(M).$$

5 Optimizations of Kid Programs

5.1 Optimizations as Rewrite rules ($\mathbf{R_{opt}}$)

Following is a partial list of optimization rules or source-to-source transformations that can be expressed as rules in the CRS for Kid. Optimizations should be performed after type checking and after all bound variables have been assigned unique names. A strategy for applying optimizations is discussed in the next section. Applicability of certain rules requires some semantic check such as "$\underline{n} > 1$". We write such semantic predicates above the line but following an "&".

$\mathbf{R_{opt}}$ is as follows:

$$\mathbf{R_{opt}} = \mathbf{R_{Kid}} \cup \mathbf{R_{oi}} \cup \mathbf{R_{io}}$$

where $\mathbf{R_{oi}}$ and $\mathbf{R_{io}}$ are defined below.

$\mathbf{R_{Kid}}$

All Kid rewrite rules, except the Cons-rule, can be applied at compile time. However, care needs to be excercised in applying some rules such as the *Application rule,* because these can cause non-termination. A similar problem exists with WLoop and FLoop. We will assume that the user will *annotate* the program to indicate when these rules are safely applicable.

$\mathbf{R_{oi}}$:

$\mathbf{R_{oi}}$ rules are applied "*outside in*", as will be shown in the next section.

Inline Substitution

$$\frac{F = \underline{\lambda_{n,m}}\,(\vec{Z_n})\,.\,(E)}{\mathsf{Ap}_{n,m}\,(F,\, \vec{X_n}) \;\longrightarrow\; (\mathcal{RB}\,[\![E]\!])\,[\vec{X_n}\,/\,\vec{Z_n}]}$$

An underlined λ corresponds to the Defsubst annotation in Id. It indicates that the function is substitutable at compiler time. Notice, if the user underlines a recursive procedure, the above rule will cause non-termination. This mechanism of underlining is often used in term rewriting systems to turn a set of rules into an equivalent strongly normalizing set of rules [12]. This mechanism also allows us to control the applicability of the loop rules. If the λ's corresponding to predicate (P) and loop body (B) of a loop expression are *not underlined*, then the loop rule can not be applicable more than once. If the user wants to unfold loops more than once at compiler time then, he must annotate the loop to indicate this. When a loop in Id is annotated to be unfolded, then the translation from Id to Kid will generate underlined λ's for P and B.

Partial Evaluation

$$\frac{F = \underline{\lambda_{\mathsf{n,m}}}\ (\vec{Z_n})\ .\ (E)}{\mathsf{Apply}_{pe}\ (F, X) \longrightarrow \{f = \lambda_{\mathsf{n-1,m}}\ (\overrightarrow{z_{n-1}})\ .\ ((\mathcal{RB}\ [\![E]\!])\ [\overrightarrow{z_{n-1}}\ /\ \overrightarrow{Z_{2,n}},\ X/Z_1])\ \ \mathsf{In}\ f\}}$$

$$\frac{F_i = \mathsf{Apply_i}\ (F,\ \underline{n},\ \vec{X_i})\ \wedge\ F = \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (E)}{\mathsf{Apply}_{pe}\ (F_i,\ X) \longrightarrow \{f = \lambda_{\mathsf{n-i-1,m}}\ (\overrightarrow{z_{n-i-1}})\ .\ ((\mathcal{RB}\ [\![E]\!])\ [\overrightarrow{z_{n-i-1}}\ /\ \overrightarrow{Z_{i+2,n}},\ \overrightarrow{X_{i+1}}\ /\ \overrightarrow{Z_{i+1}}])\ \ \mathsf{In}\ f\}}$$

A similar rule applies for $\underline{\lambda_{\mathsf{n,m}}}$.

Apply_{pe} indicates an Apply which the user has annotated for partial evaluation. Notice the difference with the previous rule. Instead of underlining the λ, the information is associated with the application. The rational being that we want to avoid the generation of too many specialized functions. This rule can also cause non termination, if an application in a recursive definition is annotated for partial evaluation. A more sophisticated strategy, that we have not explored yet, would try to compute some fixpoint in case of such an annotation.

Fetch Elimination

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\mathsf{Cons\_1}\ (X) \longrightarrow X_1}$$

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\mathsf{Cons\_2}\ (X) \longrightarrow X_2}$$

$$\frac{X = \mathsf{Cons}\ (X_1,\ X_2)}{\mathsf{List\_case}\ (X,\ E_1,\ E_2) \longrightarrow E_2}$$

The above rules are useful because we do not reduce $\mathsf{Cons}\ (X_1,\ X_2)$ into $\mathsf{Open\_cons}$ at compile time.

Algebraic Identities

$\mathbf{Alg_1}$	$\mathbf{Alg_2}$	$\mathbf{Alg_3}$
$\mathsf{And}\ (\mathsf{True},\ X) \longrightarrow X$	$\mathsf{And}\ (\mathsf{False},\ X) \longrightarrow \mathsf{False}$	$\dfrac{X = +\ (X_1, \underline{m})\quad \&\ \underline{m} > 0}{\mathsf{Less}\ (X_1,\ X) \longrightarrow \mathsf{True}}$
$\mathsf{Or}\ (\mathsf{False},\ X) \longrightarrow X$	$\mathsf{Or}\ (\mathsf{True},\ X) \longrightarrow \mathsf{True}$	$\dfrac{X = +\ (X_1, \underline{m})\quad \&\ \underline{m} > 0}{\mathsf{Less}\ (X,\ X_1) \longrightarrow \mathsf{False}}$
$+\ (X, 0) \longrightarrow X$	$*\ (X, 0) \longrightarrow 0$	$\dfrac{X = +\ (X_1, \underline{m})\quad \&\ \underline{m} > 0}{\mathsf{Greater}\ (X_1, X) \longrightarrow \mathsf{False}}$
$*\ (X, 1) \longrightarrow X$	$-\ (X, X) \longrightarrow 0$	$\dfrac{X = +\ (X_1, \underline{m})\quad \&\ \underline{m} > 0}{\mathsf{Equal?}\ (X_1,\ X) \longrightarrow \mathsf{False}}$
$\vdots$	$\mathsf{Equal?}\ (X,\ X) \longrightarrow \mathsf{True}$	$\vdots$
	$\vdots$	

Eliminating Circulating Variables

Suppose in the Id loop body of an Id program there exists an expression like "`Next x = x`", then the variable x can be made into a free variable of the loop and its circulation can be avoided. Without loss of generality we assume that the nextified variable to be eliminated is the last one.

$$\frac{\begin{array}{l} P = \lambda_{\mathsf{n}}\,(\vec{X_n})\,.\,(E) \\ B = \lambda_{\mathsf{n,n}}\,(\vec{X'_n})\,.\,(\{_{\mathsf{n}}\; S \;\mathsf{In}\;\; \vec{Z_{n-1}},\; X'_n\}) \end{array}}{\begin{array}{l} \mathsf{WLoop_n}\,(P,\; B,\; \vec{Y_n},\; Y_p) \;\longrightarrow \\ \qquad \{\; p = \lambda_{\mathsf{n-1}}\,(\vec{x_{n-1}})\,.\,(\mathcal{RB}[\![E]\!]\,[\vec{x_{n-1}}\,/\,\vec{X_{n-1}},\; Y_n/X_n]); \\ \qquad\;\; b = \lambda_{\mathsf{n-1,n-1}}\,(\vec{x'_{n-1}})\,.\,(\mathcal{RB}[\![\{_{\mathsf{n-1}}\; S \;\mathsf{In}\;\; \vec{Z_{n-1}}\}]\!]\,[\vec{x'_{n-1}}\,/\,\vec{X_{n-1}},\; Y_n/X'_n]); \\ \qquad\;\; \vec{t_{n-1}} = \mathsf{WLoop_{n-1}}\,(p,\; b,\; \vec{Y_{n-1}},\; Y_p) \\ \qquad\;\;\; \mathsf{In}\;\; \vec{t_{n-1}},\; Y_n\} \end{array}}$$

A similar optimization applies to for-loops.

Eliminating Circulating Constants

Suppose in the loop body there exists an expression like "`Next x = t`", where the variable t is a free variable of the loop body then its circulation can be avoided. Such situations may arise as a consequence of lifting invariants from a loop. The following example illustrates this transformation:

```
{ While (p x y) do
    Next x = t;
    Next y = f x y;
  Finally  y}
```

This may be transformed as follows:

```
If (p x y) then
      { y1 = f x y;
       In
         { While (p t y1) do
             Next y1 = f t y1;
          Finally  y1} }
         else
      y
```

Notice that it is only after the first iteration that the value of "x" is t. Thus, to avoid the circulation of the nextified variable "x", the loop has to be peeled once. This rule can be expressed as follows. Please note that we could have also written Z_n instead of t_n on the right-hand-side.

$$\frac{\begin{array}{l} P = \lambda\mathsf{n}\ (\vec{X_n})\,.\,(E) \\ B = \underbrace{\lambda\mathsf{n,n}\ (\vec{X'_n})\,.\,(\{\mathsf{n}\ S\ \mathsf{In}\ \vec{Z_n}\})}_{\rho}\ \ \&\ \ FE(Z_n, \rho) \end{array}}{\begin{array}{l} \mathsf{WLoopn}\ (P,\ B,\ \vec{Y_n},\ Y_p) \quad \longrightarrow \\ \qquad \mathsf{Bool\_casen}\ (Y_p, \\ \qquad\qquad \begin{array}{lcl} \{\mathsf{n}\ p & = & \lambda_{\mathsf{n-1}}\ (\vec{x_{n-1}})\,.\,(\mathcal{RB}[\![E]\!]\,[\vec{x_{n-1}}\ /\ \vec{X_{n-1}},\ t_n/X_n]); \\ \quad b & = & \lambda_{\mathsf{n-1,n}}\ (\vec{x'_{n-1}})\,.\,(\mathcal{RB}[\![\{_{\mathsf{n-1}}\ S\ \mathsf{In}\ \vec{Z_{n-1}}\}]\!]\,[\vec{x'_{n-1}}\ /\ \vec{X'_{n-1}},\ t_n/X'_n]); \\ \quad \vec{t_n} & = & \mathsf{Apn,n}\ (B,\ \vec{Y_n}); \\ \quad t_p & = & \mathsf{Ap}_{\mathsf{n-1}}\ (p,\ \vec{t_{n-1}}); \\ \quad \vec{t'_{n-1}} & = & \mathsf{WLoop}_{\mathsf{n-1}}\ (p,\ b,\ \vec{t_{n-1}},\ t_p); \\ \quad\ \mathsf{In}\ \vec{t'_{n-1}},\ t_n\}, & & \\ \vec{Y_n}) & & \end{array} \end{array}}$$

Peeling the Loop

$$\mathsf{FLoopn}\ (U,\ D,\ B,\ \vec{X_n},\ X) \quad \longrightarrow \quad \mathsf{Bool\_casen}\ (X,\ \begin{array}{lcl} \{\mathsf{n}\ \vec{t_{2,n}} & = & \mathsf{Ap}_{\mathsf{n,n-1}}\ (B,\ \vec{X_n}); \\ \quad t_1 & = & +\ (X_1,\ D); \\ \quad t_p & = & <\ (t_1,\ U); \\ \quad \vec{t'_n} & = & \mathsf{FLoopn}\ (U,\ D,\ B,\ \vec{t_n},\ t_p) \\ \quad\ \mathsf{In}\ \vec{t'_n}\}, & & \\ \vec{X_n}) & & \end{array}$$

Notice that the above rule is again applicable to the Floop generated on the RHS. To avoid unbounded number of applications of this rule the user will have to indicate how many times the loop peeling should be performed.

Loop Body Unrolling K times

$$\frac{\& \ \text{remainder}\ ((U - X_1)/D, \underline{k}) = 0}{\begin{array}{llll} \mathsf{FLoopn}\ (U,\ D,\ B,\ \vec{X_n}, X_p) & \longrightarrow & \{\mathsf{n}\ \ b = \lambda_{\mathsf{n,n-1}}(\vec{x_n})\ .\ (\{\mathsf{n-1} & \begin{array}{lll} \vec{t^1_{2,n}} & = & \mathsf{Ap_{n,n-1}}\ (B,\ \vec{x_n}); \\ t^1_1 & = & +\ (X_1,\ D); \\ \vec{t^2_{2,n}} & = & \mathsf{Ap_{n,n-1}}\ (B,\ \vec{t^1_n}); \\ t^2_1 & = & +\ (t^1_1,\ D); \\ \vdots & & \\ \vec{t^k_{2,n}} & = & \mathsf{Ap_{n,n-1}}\ (B,\ \vec{t^{k-1}_n}) \\ \multicolumn{3}{l}{\mathsf{In}\ \vec{t^k_{2,n}}\});} \end{array} \\ & & \quad \vec{t_n} = \mathsf{FLoopn}\ (U,\ D,\ b,\ \vec{X_n},\ X_p); & \\ & & \quad \mathsf{In}\ \vec{t_n}\} & \end{array}}$$

Suppose $r = \mathit{remainder}((U - X_1)/D, \underline{k})$, and r is not zero. We can still apply the above transformation by first peeling the loop r times. Notice, k has to be supplied by the user.

$\mathbf{R_{io}}$:

These optimizations need to be applied "inside-out", because that way bigger and bigger common expressions or expressions to be lifted can be formed.

Common Subexpression Elimination

$$\frac{\vec{Y_m} = PFN_m\ (\vec{X_n})}{PFN_m\ (\vec{X_n}) \ \longrightarrow\ \vec{Y_m}}$$

Primitive functions I_array, Open_cons, Apply, $\mathsf{Ap_{n,m}}$, WLoop, and FLoop are excluded from this optimization because they (may) cause side-effects.

Lift Free Expressions

$$\frac{\& \ FE(E, \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (\{\mathsf{m}\ Y = E;\ S\ \mathsf{In}\ \vec{X_m}\}))}{\begin{array}{lll} \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (\{\mathsf{m}\ Y = E;\ S\ \mathsf{In}\ \vec{X_m}\}) & \longrightarrow & \{\mathsf{m}\ \ t_1 = E; \\ & & \quad\ \ t = \lambda_{\mathsf{n,m}}\ (\vec{Z_n})\ .\ (\{\mathsf{m}\ Y = t_1;\ S\ \mathsf{In}\ \vec{X_m}\}) \\ & & \quad \mathsf{In}\ t\,\} \end{array}}$$

Where $FE(e, e')$ return true if the expression e is free in e'. This optimization allows us to deal with loop invariants, that is, expressions that do not depend on the nextified variables. A similar rule applies for $\underline{\lambda_{\mathsf{n,m}}}$. (See the restrictions in the common subexpression elimination rule). This rule in conjunction with the loop rules will cause loop invariants to be lifted from loops.

Hoisting Code out of a Conditional

$$\frac{\&\ FE(E, (\mathsf{Bool\_casen}\ (X,\ \{\mathsf{n}\ Y = E;\ S\ \ \mathsf{In}\ \ \vec{X_n}\ \},\ \{\mathsf{n}\ Y' = E;\ S'\ \ \mathsf{In}\ \ \vec{X'_n}\ \})))}{\mathsf{Bool\_casen}\ (X,\ \{\mathsf{n}\ Y = E;\ S\ \ \mathsf{In}\ \ \vec{X_n}\ \},\ \{\mathsf{n}\ Y' = E;\ S'\ \ \mathsf{In}\ \ \vec{X'_n}\ \})\ \longrightarrow}$$

$$\begin{array}{lll} \{\mathsf{n}\ t_1 & = & E; \\ \quad \vec{t_n} & = & \mathsf{Bool\_casen}\ (X, \\ & & \quad \{\mathsf{n}\ Y = t_1;\ S\ \ \mathsf{In}\ \ \vec{X_n}\ \}, \\ & & \quad \{\mathsf{n}\ Y' = t_1; S'\ \ \mathsf{In}\ \ \vec{X'_n}\}) \\ \quad \mathsf{In}\ \vec{t_n}\ \} & & \end{array}$$

5.2 A Partial Evaluator for Kid

Applying $\mathbf{R_{kid}}$ rules and $\mathbf{R_{oi}}$ rules is like *partial evaluation* of a Kid program. In the following we present an efficient strategy for applying $\mathbf{R_{kid}}$ and $\mathbf{R_{oi}}$ rules using the function $\mathcal{EV}$. However, space does not permit us to give a full interpreter. Therefore, we illustrate our strategy using a few carefully chosen rule, *i.e.* $R_{CRS}\ \cup\ \{\delta$ rules, *Case* rules for booleans, P_select rule $\}$. The extension of $\mathcal{EV}$ to other rules of $\mathbf{R_{kid}}$ and $\mathbf{R_{oi}}$ is straightforward. As usual, initially ρ and σ would be empty.

$\mathcal{EV}[\![+\ (\underline{m},\ \underline{n})]\!]\ \rho\ \sigma = \underline{+(m,\ n)}$

$\mathcal{EV}[\![\mathsf{Bool\_casem}\ (\mathsf{True},\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{EV}[\![E_1]\!]\ \rho\ \sigma$

$\mathcal{EV}[\![\mathsf{Bool\_casem}\ (\mathsf{False},\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathcal{EV}[\![E_2]\!]\ \rho\ \sigma$

$\mathcal{EV}[\![\mathsf{Bool\_casem}\ (X,\ E_1,\ E_2)]\!]\ \rho\ \sigma = \mathsf{Bool\_casem}\ (X,\ \mathcal{EV}[\![E_1]\!]\ \rho\ \sigma,\ \mathcal{EV}[\![E_2]\!]\ \rho\ \sigma])$

$\mathcal{EV}[\![\text{P\_select}\ (X,\ Y)]\!]\ \rho\ \sigma = Z \qquad \text{if } (\mathcal{L}\ (X, \sigma)) = \text{P\_store}\ (X,\ Y,\ Z)$

$\mathcal{EV}[\![\{Ss;\ As;\ Bs\ \text{In}\ X\}]\!]\ \rho\ \sigma = e$

where e is obtained by the execution of the following program written in pseudo-Id

```
{ blk, - = F&S({Ss; As; Bs In X});
  flag = True;
  If blk = ⊤ then ⊤
   else
  { blk = { While flag do
              Suppose blk is {Ss; As; Bs In X}
             is = CC(σ ∪ Ss);
             fs = ρ ∪ As;
             next blk, next flag =  If is = ⊤s then (⊤, False)
                                      else F&S(EV'_RHS(blk, is, fs))
             finally blk}
    Suppose blk' is {Ss'; As;' Bs' In X'}
    is = CC(σ ∪ Ss);
    fs = ρ ∪ As;
    blk'' = If is = ⊤s then (⊤, False)
              else (EV_RHS(blk'', is, fs))
    In blk''}}
```

where

$$\mathcal{EV}'\_\mathcal{RHS}([\![Ss;\ As;\ X_1 = E_1; \cdots X_n = E_n\ \text{In}\ X_i]\!]\ \rho\ \sigma =$$
$$\{Ss;\ As;\ X_1 = \mathcal{EV}'(E_1,\ \rho\ \sigma); \cdots X_n = \mathcal{EV}'(E_n,\ \rho\ \sigma)\ \text{In}\ X_i\}$$

and the function $\mathcal{EV}'$ given below unlike $\mathcal{EV}$ does not go inside encapsulators.

$\mathcal{EV}'[\![+\ (\underline{m},\ \underline{n})]\!]\ \rho\ \sigma = \underline{+(m,\ n)}$

$\mathcal{EV}'[\![\text{Bool\_casem}\ (\text{True},\ E_1,\ E_2)]\!]\ \rho\ \sigma = E_1$

$\mathcal{EV}'[\![\text{Bool\_casem}\ (\text{False},\ E_1,\ E_2)]\!]\ \rho\ \sigma = E_2$

$\mathcal{EV}'[\![\text{P\_select}\ (X,\ Y)]\!]\ \rho\ \sigma = Z,\ \text{True} \qquad \text{if } (\mathcal{L}\ (\ X, \sigma)) = \text{P\_store}\ (X,\ Y,\ Z)$

else

$\mathcal{EV}'[\![E]\!]\ \rho\ \sigma = E$

else

$\mathcal{EV}[\![E]\!]\ \rho\ \sigma = E$

Theorem 5.1 (Normalization Theorem) *Given terms M and N, and let $R = R_{CRS} \cup R_{kid} \cup R_{oi}$*

$$M \underset{R}{\twoheadrightarrow} N \implies \mathcal{EV}[\![M]\!]\ \text{Nil Nil} = N.$$

We do not describe the strategy for $\mathbf{R_{io}}$ rules because it requires choosing a data structure for Kid programs. These rules should be applied from inside out. Unfortunately $\mathbf{R_{io}}$ rules can trigger some $\mathbf{R_{oi}}$

optimizations. For example, the cse-rule may trigger an algebraic rule, such as, the equality rule.

Briefly the overall strategy for optimizations is as follows:

(1) apply $\mathbf{R}_{\mathbf{Kid}} \cup \mathbf{R}_{\mathbf{oi}}/\mathsf{Cons}$ rule, from outside in, as explained in Section 5.2;
(2) apply $\mathbf{R}_{\mathbf{io}}$ rules from inside out;
(3) if there is any change in the expression in step (2) go to step (1), else stop.
We beleive that the above strategy is normalizing with respect to $\mathbf{R}_{\mathbf{opt}}$.

An other interesting question about optimizations is whether $\mathbf{R}_{\mathbf{opt}}$ is confluent. We have shown in [1] that $\mathbf{Alg_3}$ rules cause $\mathbf{R}_{\mathbf{opt}}$ to be non confluent. However, this is not a serious drawback because the cases where the confluence is lost are the ones where the unoptimized program would have produced no information .

5.3 Correctness of Optimizations

Definition 5.2 (Observational Equivalence) *Given two terms M and N, M and N are said to be Observationally Equivalent iff*

$$\mathcal{P}rint^* \ (M) \equiv \mathcal{P}rint^* \ (N).$$

For correctness, observational equivalence is not enough. It has to be shown that no context can distinguish between optimized and unoptimized term.

Definition 5.3 (Observational Congruence) *Given two terms M and N, M and N are said to be Observationally Congruent iff*

$$\forall\, C[\square],\ \ \mathcal{P}rint^* \ (C[M]) \equiv \mathcal{P}rint^* \ (C[N])\,.$$

Definition 5.4 (Correctness) *An optimizer (A,R_{opt}) of (A,R) is correct iff*

$$\forall\, M \in A, M \twoheadrightarrow_{R_{opt}} N \Longrightarrow \begin{array}{l} (i)\ \ \mathcal{P}rint^*(M) \sqsubseteq \mathcal{P}rint^*(N) \ \textit{and} \\ (ii)\ \ \mathcal{P}rint^*(N) = \top \Longrightarrow \mathcal{P}rint^*(M) = \top. \end{array}$$

Notice that condition *(ii)* is needed because $\top$ is higher than all print values and we do not want the optimizer to take all programs to $\top$.

It is believed that all optimizations presented in Section 5.1 preserve correctness, though this has been proven for only a small subset of them so far [1]. In general, correctness of an optimization is difficult to prove. However, some optimizations can be proven correct easily because they are *derived* rules.

Definition 5.5 (Derived rule) *A rule r is said to be derived in R iff*

$$M \xrightarrow{r} M_1 \Longrightarrow \exists\, M_2,\ \ M \twoheadrightarrow_{R} M_2 \ \wedge\ M_1 \twoheadrightarrow_{R} M_2$$

Corollary 5.6 *Given an optimizer $\mathcal{O}$ (A,R_{opt}) of (A,R),*

$$\forall\, r \in R, r \text{ is derived } \implies \mathcal{O} \text{ is correct}.$$

Lemma 5.7 *All* $\mathbf{R_{Kid}}$ *rules, Inline Substitution and Fetch Elimination are derived rules, and hence correct.*

Unfortunately, it is not always possible to mimic inside R_{Kid} what an optimizations does. Therefore, we introduce a notion of *graph equivalence*, which is useful for proving the correctness of *cse*-like rules. The terms of a graph may be described by the grammar of Figure 1 by changing the definition of the SE syntactic category with $SE = E$. Graphs for a Kid term can be generated by applying a new substitution rule, called the E_substitution rule.

E_substitution rule

$$\frac{X = E}{X \longrightarrow E}$$

where E can be any *side-effect free expression*, (sef in short), as defined below.

Definition 5.8 (Side-effect-free expression) *(1) Variables and constants are sef;*
(2) all $PF_{n,m}\ (\vec{X_n})$ *, except* I_array, Open_cons, Apply, WLoop and FLoop, *are sef;*
(3) Bool_Case$_m$ $(X,\ E_1,\ E_2)$ *is sef if* E_1 *and* E_2 *are sef; similarly for* List_Case.
(4) $\lambda_{n,m} x.E$ *is sef if* E *is sef;*
(5) $Ap_{n,m}$ $(F,\ X)$ *is sef if* F *is bound to a sef* λ*-abstraction;*
(6) WLoop$_m$ $(P, B, \vec{X_n},\ Y)$ *is sef if* P *and* B *are bound to sef* λ*-abstractions; similarly for* FLoop.

Suppose $\mathbf{R'_{CRS}}$ be {E_substitution rule, Block flattening rule, Multivariable rule}.

Definition 5.9 (Unravelled form) *Let N be the normal form of a term M with respect to R'_{CRS}. U(M), the unravelling form of a term M, is the term obtained by deleting from N all bindings of the form $X = E$ (where E is a side-effect free expression).*

Notice that the unravelled version of a term is not necessarily a tree.

Definition 5.10 (graph-equivalence $\equiv_g$) *Two terms M and N are said to be graph-equivalent, if*

$$U(M)\ \equiv_\alpha\ U(N).$$

Corollary 5.11 *Given an optimizer $\mathcal{O}$ (A,R_{opt}) of (A,R),*

$$\forall\, M,\ N \in A,\ [M \underset{opt}{\longrightarrow} N \implies M \equiv_g N] \implies \mathcal{O} \text{ is correct}.$$

Lemma 5.12 *Let R be the set {Common subexpression elimination rule, lift free expressions rule, Hoisting code out of a conditional rule }.*

$$M \underset{R}{\longrightarrow} N \implies M \equiv_g N.$$

Hence *cse*-like rules are correct.

6 Conclusions

Kid goes a long way towards expressing many machine independent implementation concerns. Since Kid has a proper calculus associated with it, correctness issues can be handled at an abstract level. Kid is central to the current restructuring of the Id compiler which is already being used by at least 4 or 5 groups to generate code for their machines by just changing back end of the compiler. The major deficiency of Kid is the inability to express storage reuse. We believe a language with a proper calculus which does not obscure parallelism and which can express storage reuse would be extremely useful in practical compilers for functional and other declarative languages.

Some of the important optimizations that we currently perform in the Id compiler but have not included in R_{opt} are *dead code elimination*, *loop variable induction* to hoist array bound checking, and *array subscript analyses*. These optimizations do not fit in the CRS model very well. A better formalization of these optimizations would be very useful.

References

[1] Z. M. Ariola and Arvind. P-TAC: A parallel intermediate language. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, London*, 1989. Also: CSG Memo 295, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.

[2] Z. M. Ariola and Arvind. Compilation of Id$^-$: a subset of Id. Technical Report CSG Memo 315, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990. Revised 1 November.

[3] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, 11(4), October 1989. An earlier version appeared in Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, USA, September/October 1986, Springer-Verlag LNCS 279, pages 336-369.

[4] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, 1984.

[5] H. P. Barendregt, B. T.H., van Eekelen M.C.J.D, G. J.R.W., K. J.R., van Leer M.O., P. M.J., and S. M. Ronan. Towards an intermediate language based on graph rewriting. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands, Springer-Verlag LNCS 259*, June 1987.

[6] T. Brus, M. van Eekelen, M. vam Leer, and M. Plasmeijer. Clean - a language for functional graph rewriting. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Portland, Oregon,Springer-Verlag LNCS 274*, 1987.

[7] B. Courcelle. *Fundamentals Properties of Infinite Trees.* D. Reidel Publishing Company, 1982.

[8] N. de Bruijn. Lambda calculus notation with nameless dummies: A tool for automatic formula manipulation, with application to the church-rosser theorem. In *Koninkijke Nederlandse Akademie van Wetenschappen, Series A, Mathematical Sciences*, 1972.

[9] P. Hudak and P. Wadler. Report on the programming language Haskell, a non-strict purely functional language (Version 1.0). Technical Report YALEU/DCS/RR777, Yale University, Department of Computer Science, April 1990.

[10] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69. Association for Computing Machinery, June 1984.

[11] R. Kennaway. Implementing term rewrite languages in dactl. *Theoretical Computer Science*, 1, 1990.

[12] J. Klop. Term rewriting systems. Course Notes, Summer course organized by Corrado Boehm, Ustica,Italy, September 1985.

[13] R. S. Nikhil. Id (Version 90.0) Reference Manual. Technical Report CSG Memo 284-a, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA, July 1990.

[14] G. Plotkin. Call-by-name, Call-by-value and the Lambda Calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[15] P. J. Simon L. *The implementation of Functional Programming Languages.* Prentice-Hall International, Englewood Cliffs, N.J., 1987.

[16] D. A. Turner. A New Implemetation Technique for Applicative Languages. In *Software - Practice and Experience*, volume 9, pages 31–49, 1979.

[17] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *IFIP Int'l Conf. on Functional Programming and Computer Architecture, Nancy, Lecture Notes in Computer Science 201, New York, 1985. Springer-Verlag*, 1985.

[18] C. Wadsworth. *Semantics And Pragmatics Of The Lambda-Calculus.* Ph.D. thesis, University of Oxford, Semtember 1971.